Comput Econ (2016) 48:253–280
DOI 10.1007/s10614-015-9509-4

Adapting and Optimizing the Systemic Model of Banking Originated Losses (SYMBOL) Tool to the Multi-core Architecture

Ronal Muresano[1] · **Andrea Pagano**[1]

Accepted: 3 August 2015 / Published online: 9 August 2015

Abstract Currently, multi-core system is a predominant architecture in the computational word. This gives new possibilities to speedup statistical and numerical simulations, but it also introduce many challenges we need to deal with. In order to improve the performance metrics, we need to consider different key points as: core communications, data locality, dependencies, memory size, etc. This paper describes a series of optimization steps done on the SYMBOL model meant to enhance its performance and scalability. SYMBOL is a micro-funded statistical tool which analyses the consequences of bank failures, taking into account the available safety nets, such as deposit guarantee schemes or resolution funds. However, this tool, in its original version, has some computational weakness, because its execution time grows considerably, when we request to run with large input data (e.g. large banking systems) or if we wish to scale up the value of the stopping criterium, i.e. the number of default scenarios to be considered. Our intention is to develop a tool (extendable to other model having similar characteristics) where a set of serial (e.g. deleting redundancies, loop enrolling, etc.) and parallel strategies (e.g. OpenMP, and GPU programming) come together to obtain shorter execution time and scalability. The tool uses automatic configuration to make the best use of available resources on the basis of the characteristics of the input datasets. Experimental results, done varying the size of the input dataset and the stopping criterium, show a considerable improvement one can obtain by using

✉ Ronal Muresano
ronal.muresano@jrc.ec.europa.eu; ronal.muresano@jrc.europa.ec.eu

Andrea Pagano
andrea.pagano@jrc.europa.ec.eu

[1] European Commission, Joint Research Centre (JRC), Institute for the Protection and the Security of the Citizen (IPSC), Financial and Economic Analysis Unit, Via E. Fermi, 2749, CP. 21027, Ispra, Italy

the new tool, with execution time reduction up to 96 % of with respect to the original serial version.

Keywords Optimization algorithm · Parallel techniques · OpenMP · GPU · SYMBOL

1 Introduction

Nowadays scientific applications are called to provide more and more accurate results which are going to be used in very different fields: engineering, economic and environmental studies, policy making, etc. However, such accuracy requires high computational power to be executed. Additionally, the current trend in computer system is to use multi-core architectures with a large number of cores. This opens new possibilities and, at the same time, new challenges, especially when we wish to fine tune an application to get the most out of the available IT infrastructure. Moreover, multi-core architecture have opened the road to include more parallelism within nodes. Lately, GPUs technology has collected an increase interest as computational efficient processors.

The first goal is to correctly identify the parallelizable sections in the code in order to take advantage of this architecture.

Whenever an application is developed in serial and we would parallelize it, we have to consider diverse key points such as: core communications, data locality, dependencies, memory size, etc., in order to improve the performance metrics. Then, to obtain the most out of multi-core capacities so to improve the performance metrics of specific applications, it is important to develop a set of *best practices* in order to manage the inefficiencies generated by the overhead added by the parallel library (Michailidis and Margaritis 2012).

This paper describes a set of adaptation techniques and the optimizations processes done on the SYMBOL model original code, aiming to improve its performance in two directions: execution time and scalability. SYMBOL is a statistical tool estimating losses deriving from bank defaults, explicitly linking Basel capital requirements to the other key tools of the banking safety net, i.e. Deposit Guarantee Schemes, and bank Resolution Funds (De Lisa et al. 2011). This tool has been used by Commission Services to prepare various Impact Assessments of European Commission (EC) regulatory proposals to enhance financial stability and prevent future crises (Capital Requirement Directive Proposal, Bank Recovery and Resolution Directive and Financial Transactions Tax). Moreover, SYMBOL is used to analyse the contributions of individual banks to total losses originated in the banking sector. This is an area of particular interest to policy-makers, as information on the factors determining risk contributions could be used in areas such as taxation of financial institutions (i.e. risk levies) and structural reform (e.g. too big-to-fail paradigma).

The original version of SYMBOL has been developed in serial, and it presents some computational weakness, leading to a dramatic increase in execution time, when the

model is run on a very large input data, lasting from several hours to days depending on the number of default scenarios in the simulations.[1]

Hence, we have designed a procedure to adapt SYMBOL to a multi-core environment taking advantage of the computational power benefits of this architecture. Then, we are improving the execution time of SYMBOL with two goals: executing with large data sets and scaling the number or default scenarios, by using in an efficient manner the computational resources.

The paper is structured as follows: a brief description of the SYMBOL model in Sect. 2. Section 3 describes how to do an efficient execution of SYMBOL on multi-core architecture. Section 4 presents the optimization results for the SYMBOL model. Conclusions are discussed in Sect. 5.

2 Systemic Model of Banking Originated Losses Model (SYMBOL)

The SYMBOL model simulates the distribution of losses in excess of banks' capital within a banking system (usually a country) using micro-data from banks' balance sheets by summing up, at system level, losses in excess of banks' capital of individual institutions.

Individual banks losses are generated via Monte Carlo simulations using the Basel Foundation Internal Ratings Based (FIRB) loss distribution function. This function is based on the Vasicek model (Vasicek 2002), which in broad terms extends the Merton model (Merton 1974) to a portfolio of borrowers,[2] based on an estimate of the average default probability of the portfolio of assets of individual bank. Usually, each SYMBOL simulation ends when 100,000 (stopping criterium) runs with at least one default are obtained.[3] This implies to let the model run for few millions of iterations in the case of small size countries and hundreds of thousands iterations for medium-large countries.

The model can also be run under the hypothesis that contagion can start among banks, via the interbank lending market. In this case, additional losses due to a contagion mechanism are added on top of the losses generated via Monte Carlo simulations, hence additional banks may default (see also Step 4 below). The model can take into

[1] This number is equivalent to set a simulation's stopping criterium. The larger is this number the smaller is the variability of the results with respect to Monte Carlo draws. Results on this aspect are out of the scope of the paper, but they are available upon request.

[2] The Basel Committee permits banks a choice between two broad methodologies for calculating their capital requirements for credit risk. One alternative, the Standardized Approach, measures credit risk in a standardized manner, supported by external credit assessments. The other alternative is the Internal Rating-Based (IRB) approach which allows institutions to use their own internal rating-based measures for key drivers of credit risk as primary inputs to the capital calculation. Institutions using the Foundation IRB (FIRB) approach are allowed to determine the borrowers' probabilities of default while those using the Advanced IRB (AIRB) approach are permitted to rely on own estimates of loss given default and exposure at default. These risk measures are converted into risk weights and regulatory capital requirements by means of risk weight formulas specified by the Basel Committee. The Basel risk weight formulas for market risk is a specifically calibrated version of the Vasicek model for credit portfolio losses. On the Basel FIRB approach, one may see BCBS (2005, 2006, 2010).

[3] This is needed to guarantee stability of the tail of the simulated distribution. Increasing the stopping criterium will enhance stability at the cost of longer simulation execution time.

account the existence of a safety-net for bank recovery and resolution, where Bail-in (BiB), Deposit Guarantee Schemes (DGS) and Resolution Funds (RF) may intervene to cover losses exceeding bank capital before they can hit public finance.

Before entering into the technical details of the model, we briefly state its underlying assumptions:

- SYMBOL approximates all risks as if they were credit risk;
- SYMBOL assumes that the FIRB formula applies for all banks and adequately represents risks banks are exposed to;
- Banks in the system are correlated with a given factor (see Step 2 below).
- The only contagion channel is the interbank lending market (see Step 4 below).
- SYMBOL assumes that all events happen at the same time (i.e. there is no sequencing in the simulated events, except when contagion between banks is considered).

We continue this section by detailing steps/assumptions of the model.

2.1 Steps of SYMBOL Model

- *Step 1*: *Estimation of the implied obligors probability of default (IOPD) of the portfolio of each individual bank* SYMBOL approximates all risks as if they were credit risk and assumes that the Basel FIRB approach appropriately describes credit risk banks are exposed to. The model estimates the average IOPD of the portfolio of each individual bank using its total minimum capital requirement (MCR)[4] declared in the balance sheet by numerical inversion of the Basel FIRB formula for credit risk. Individual bank data needed to estimate the IOPD are banks MCR and total assets, which can be derived from the balance sheet. All other parameters are set to their regulatory default values.
- *Step 2*: *Simulation of correlated losses for the banks in the system* Given the estimated average IOPD, SYMBOL assumes that correlated losses hitting banks can be simulated via Monte Carlo using the same FIRB formula (BCBS 2006) and imposing a correlation structure among banks (with a correlation set to $\rho = 50\,\%$). This correlation exists either as a consequence of the banks' common exposure to the same borrower or, more generally, to a particular common influence of the business cycle.[5]

  In each simulation run j, losses for bank i are simulated as:

$$L_{i,j} = LGD \cdot N\left[\sqrt{\frac{1}{1-R}}N^{-1}(IOPD_i) + \sqrt{\frac{R}{1-R}}N^{-1}(\alpha_{i,j})\right] \quad (1)$$

[4] Banks must comply with capital requirements not only for their lending activity and credit risk component. Banks assets are in fact not only made up of loans, and there are capital requirements that derive from market risk, counter-party risk, and operational risk, etc. The main assumption currently behind SYMBOL is that all risk can be approximated as credit risk. Except for very large banks with extensive and complex trading activities, this simplifying assumption is not excessively distortive as credit risk usually accounts for a very large share of banks' total capital requirements.

[5] The choice of the 50 % correlation is based on (Sironi and Zazzara 2004). A discussion and a sensitivity check on this assumption can be found in (De Lisa et al. 2011).

where N is the normal probability function and $N^{-1}(\alpha_{i,j})$ are correlated normal random shocks, and $IOPD_i$ is the average implied obligors probability of default estimated for each bank in step 1. LGD is the Loss Given Default, set as in Basel regulation to 45 % and R is the correlation factor among individual bank assets. This R value is calculated using the Eq. 2, where PD is the obligor probability default (De Lisa et al. 2011).

$$R = 0.12 * \frac{(1 - EXP(-50 * PD))}{(1 - Exp(-50))} + 0.24 \left[1 - \frac{(1 - EXP(-50 * PD))}{(1 - Exp(-50))} \right] \quad (2)$$

– *Step 3*: *Determination of the default event* Given the matrix of correlated losses, SYMBOL determines which banks fail. As illustrated in Fig. 1, a bank default happens when simulated obligor portfolio losses $L_{(i,j)}$ exceed the sum of the expected losses $EL_{(i)}$ and the total actual capital $K_{(i)}$ given by the sum of its MCR plus the bank's excess capital (i represents the bank and j is the simulation scenario) (Eq. 3).

$$L_{(i,j)} > El_{(i)} + \mathrm{K_{(i)}} \quad (3)$$

The light-gray area in Fig. 1 represents the region where losses are covered by provisions and total capital, while the dark-gray shows when banks default under the adopted definition. The probability density function of losses for an individual bank is skewed to the right, i.e. there is a very small probability of extremely large losses and a high probability of losses that are closer to the average/expected loss. The Basel VaR is the Value at Risk corresponding to a confidence level of 0.1 %, i.e. losses from the obligors' portfolio are lower that the Basel VaR with probability 99.9 %. This percentile falls in the light-gray area as banks generally hold an excess capital buffer on top of the minimum capital requirement. Data needed for determining the default event for each bank is its level of total capital.

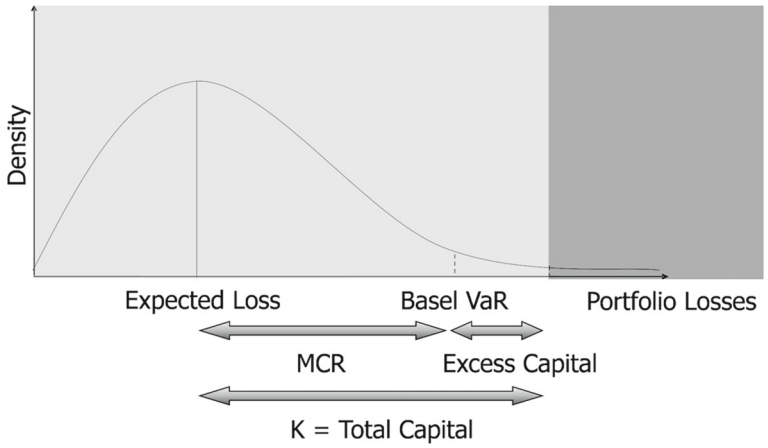

Fig. 1 Individual bank loss probability density function

– *Step 4 (Optional): contagion mechanism*. SYMBOL can include a direct contagion mechanism since the default of one bank can compromise the solvency of its creditor banks, thus triggering a domino effect in the banking system. SYMBOL focuses on the role of the interbank lending market in causing contagion. The more a bank is exposed in the interbank market, the more it will suffer from a default in the system. In fact the failure of a bank drives additional losses on the others, equal to 40 % of the amounts of its total interbank debts.
As bank-to-bank interbank lending positions are not publicly available, an approximation is needed to build the whole matrix of interbank linkages. The model assumes that each bank is linked with all others and uses a criterion of proportionality to distribute additional contagion losses: the amount of losses distributed to each bank is determined by the share of its creditor exposure in the interbank market.[6]
A default driven by contagion effects occurs whenever additional due to the contagion channel loss causes any new bank default. This contagion process stops when no new additional bank defaults.
The magnitude of contagion effects depends on the two assumptions made: first the 40 %[7] of interbank debits that are passed on as losses to creditor banks in case of failure, and, second, the criterion of proportionality used to distribute these losses across banks.[8]
– *Step 5*: *Aggregated distribution of losses for the whole system* Aggregate losses are obtained summing up losses in excess of capital plus potential recapitalisation needs[9] of all banks in the system in each simulation run. This includes both failed and non-failed banks, reflecting the fact that all banks in the system remain viable.

2.2 Computational Tool of SYMBOL

The steps discussed before can be divided in three main parts. The first one is the preprocessing of SYMBOL, steps from 2 to 4 are part of a computational tool designed in C, where the economic scenarios are simulated and losses compared with the bank's actual capital. Last, the the post processing part of the model can take into account different settings and asses the impact of foreseen reforms.

[6] In formula, if a bank j fails, losses due to contagion on bank k equal to:

$$L_k^{Contagion} = 40\,\% IB_j^- \frac{IB_k^+}{\sum_{h \neq j} IB_h^+}$$

where IB^- and IB^+ are respectively the interbank debts and credits of a bank.

[7] is coherent with the upper bound of economic research on this issue, see e.g. (James 1991; Mistrulli 2007) and (Upper and Worms 2004).

[8] Other theoretical network designs aiming to mimic the interbank markets may be used, but this aspect is beyond the scope of the present work [see (Zedda et al. 2012)].

[9] Recapitalisation needs can be set equal to the Basel minimum capital requirements [4.5 or 8 % risk weighted assets (RWA)] and are estimated also for non-defaulted banks when simulated losses erode this threshold.

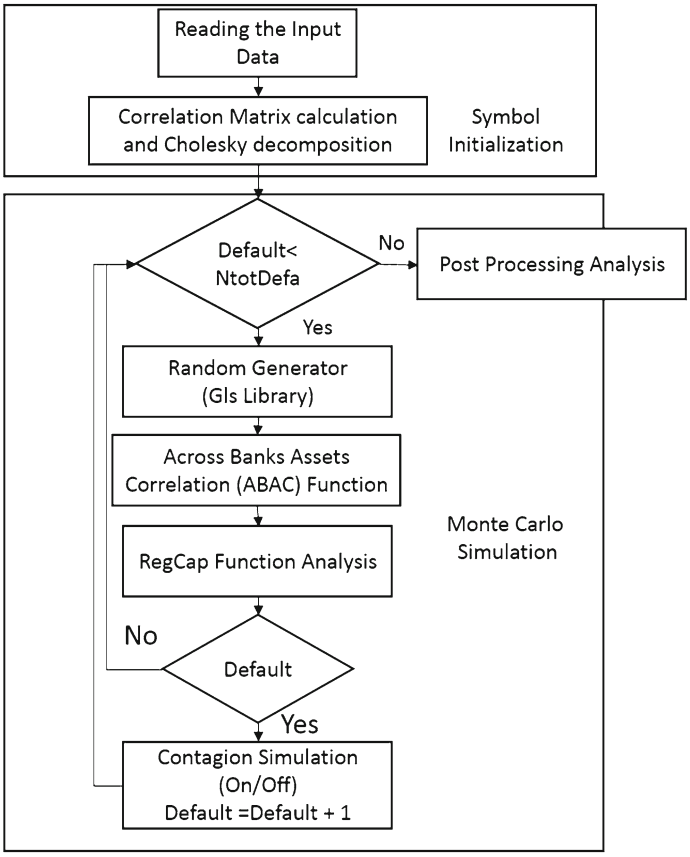

Fig. 2 Flowchart for the SYMBOL algorithm

This paper mainly focuses on the computation tool (2–4). In particular we would like to address how three key factors impact the computation burden, these factors being: level of stopping criterium, the input data (i.e. individual banks minimum capital, total capital, interbank loans and deposits), and finally, a contagion flag (contagion being active or inactive). A flowchart diagram of SYMBOL algorithm is illustrated in the Fig. 2, where we have divided the code in three main parts: (1) the initialization of the program, (2) simulation of correlated losses by Monte Carlo simulation (this part is split into three additional sub items: (2a) a random generator, (2b) an Across Banks Assets Correlation (ABAC) function (it computes the assets correlation among banks in the sample) and (2c) regulatory capital RegCap function (it calculates the Eq. 1) and (3) the last module which deals with contagion analysis. Finally the output data is kept in files to make the post-processing analysis.

When SYMBOL is executed, we have to deal with two main issues with respect to execution time: the first one is related to the computational time, which presents significant variability depending on the input data (number of banks by countries). This is detailed in Fig. 3, where we have executed SYMBOL using a set of data input

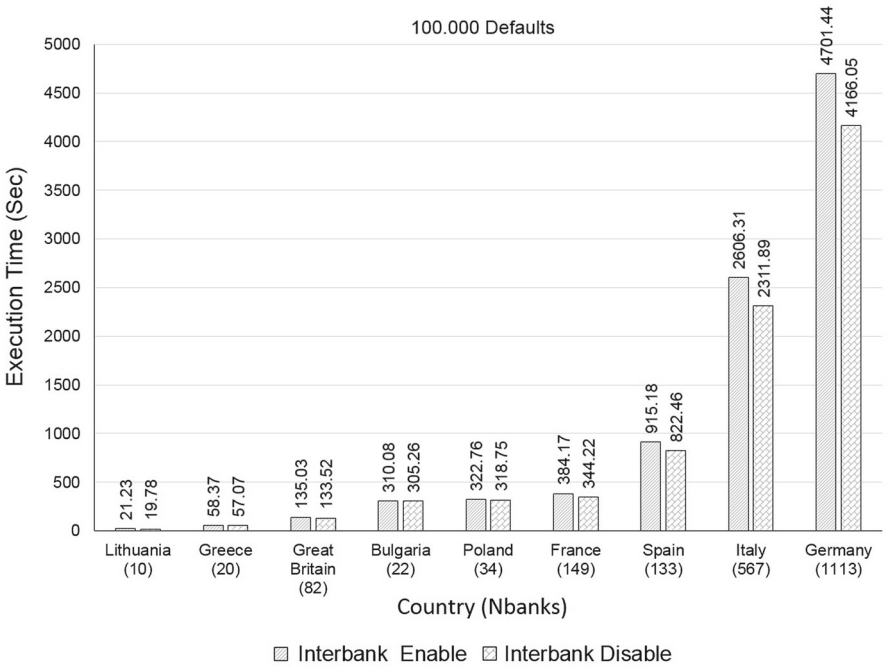

Fig. 3 Serial execution time for SYMBOL tool

from different European Union (EU) countries. A clear example is given by Lithuania input (10 banks) executed in few seconds, while Germany (1113 banks) took around 1 hour and 19 minutes. This is somehow expected, because the number of banks in the two case is very different. However, execution time will not only depend on the number of banks. If we observe execution time for France (149 banks) and Spain (133 banks), we can see that France's execution time is much lower than Spain (almost divided by 2). A similar situation occurs with Greece and Bulgaria where the number of banks is similar, but the execution time differs of almost one order of magnitude. Hence, execution time will depend on the data as a whole, not just on their size.

The second issue is linked to the number of defaults requested to stop the simulation. In this case, execution time grows almost linearly with the number of defaults. This behavior is illustrated in Fig. 4, where we have selected two large input, the first one is Germany (1113 banks) and the second is a mix between a set of banks from different EU countries (2727 banks). In addition, we can observe that, execution time is going to be measured in days, whenever the number of defaults and the number of banks are increased. For example, if we execute a simulation with 1.000.000 defaults scenarios using the mix EU banks (2727 banks), execution time is around 4 days (Fig. 4). This execution time begins to be non-viable, considering that simulations under different settings for the full EU sample may be executed frequently. This already justifies the need for parallelizing and optimizing the code.

Our approach aims to take advantage of the benefits of the computational power of multi-core architecture in order to improve the execution time with two goals:

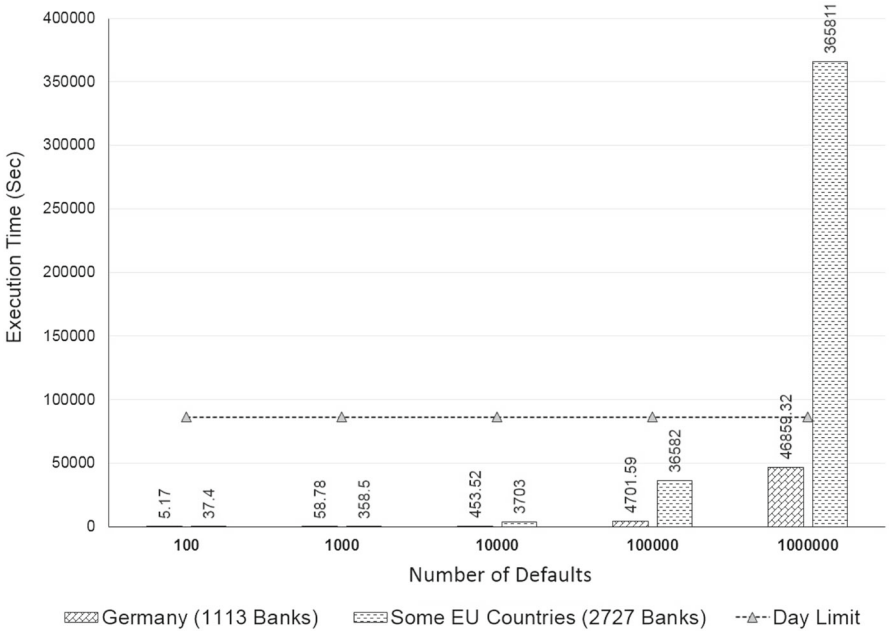

Fig. 4 SYMBOL's execution time behavior when are scaled the number of banks and default

executing with large input dataset and scaling up the number or default scenarios. However, these goals should be attained with a shorter response time (dropping the execution time), but also using in an efficient manner the computational resources. Hence we need to deeply analyse SYMBOL model structure in order to identify which parts of the code present dependencies and which parts of the code can be executed in parallel. It is important to understand that these dependencies create synchronization problems that affect the execution time in the parallel version.

3 Efficient Execution of SYMBOL on Multi-core Architecture

Designing an application that exploits the multi-core architecture presents several challenges. On the other hand, there are numerous research efforts targeting these architectures (Michailidis and Margaritis 2012; De Rose et al. 2011). In all these efforts, performance evaluation and analysis have an important role to understand the current and upcoming challenges for a better utilization of such systems.

In this sense, we have designed a method, which allows us to evaluate the characteristics of SYMBOL model in order to improve its performance (execution time and speed-up). This method can be extended to other economic models with similar behavior. Our method consists in five steps (code and model analysis, designing the optimization technique, software implementation, testing version and data verification) (see Fig. 5).

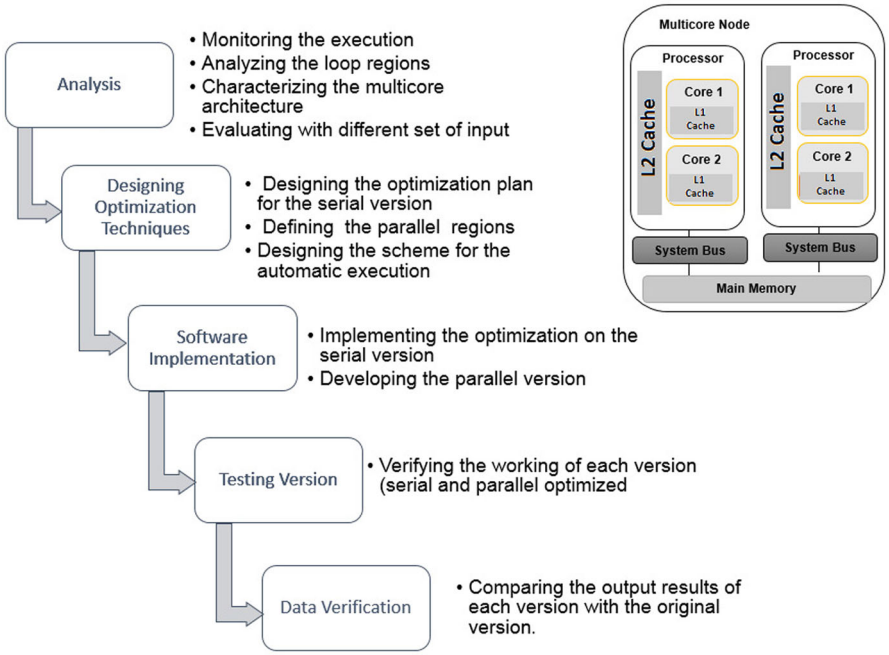

Fig. 5 Method for analyzing and improving the SYMBOL's tool performance

These steps allow us to apply the optimization and parallel techniques to create a new version improving the performance in terms of (decreasing the) execution time and enhancing the scalability of model (executing with large input data and a bigger amount of defaults). Obviously the output of the new version should be 100 % equal to the original version.

3.1 Code and Model Analysis Step

Here, we analyse SYMBOL code and the data structures. The first step is to monitor the simulation using a set of timers to analyse the execution time behavior of each part of the code. To do this, we have executed SYMBOL with a small input data (Lithuania). A summary of the results is presented in Fig. 6, where we can see the impact of each function on the execution time.

In this example, the RegCap (Regulatory Capital) function takes around 56 % of the execution, while the other two main important segments of the code (i.e. random generator and ABAC function) use 15 and 19 % respectively (Fig. 6).

Hence we get an idea about which part of the code has to be optimized first. However, considering the initial execution time as in Fig. 3, we have to evaluate the overall execution time using different sets of input data (small, medium and large) in order to know the real behavior of the model. The main results of these evaluations are shown in Fig. 7, where we can observe that execution time of each function can vary considerably depending of the input size.

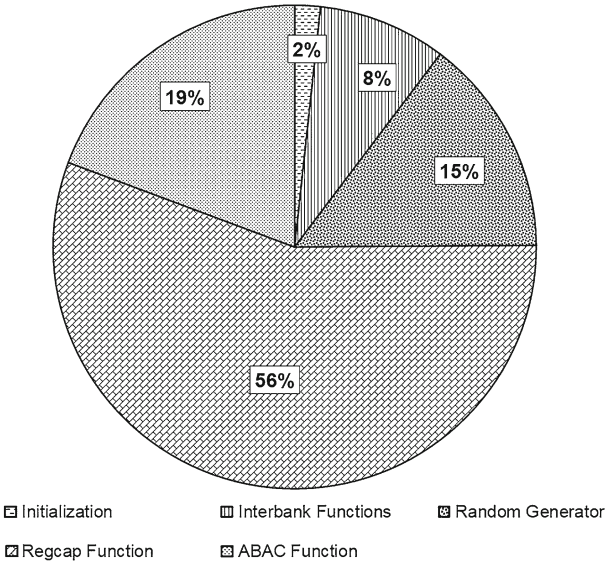

Fig. 6 Execution time modules for Lithuania input data (10 banks)

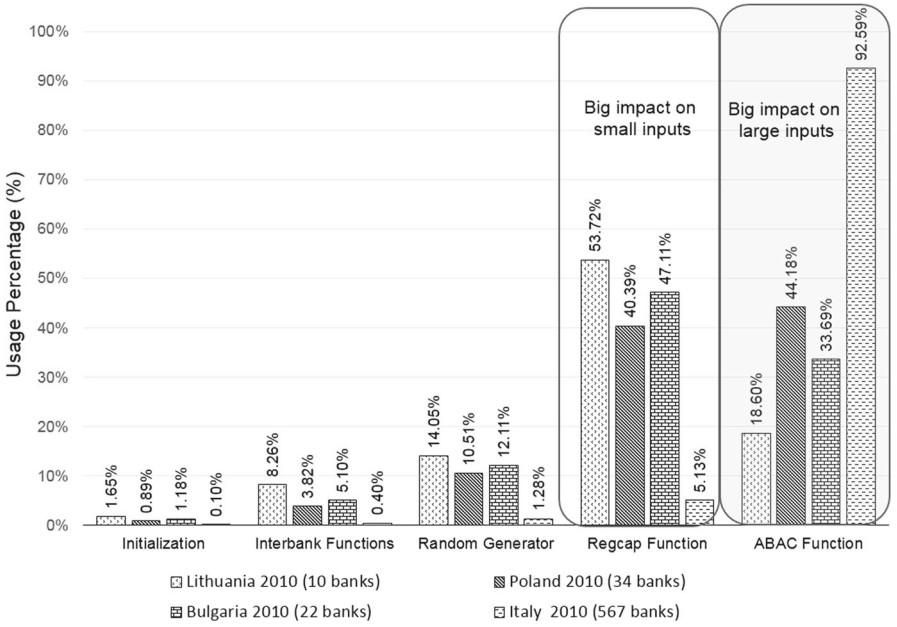

Fig. 7 SYMBOL's execution time behavior for different inputs data

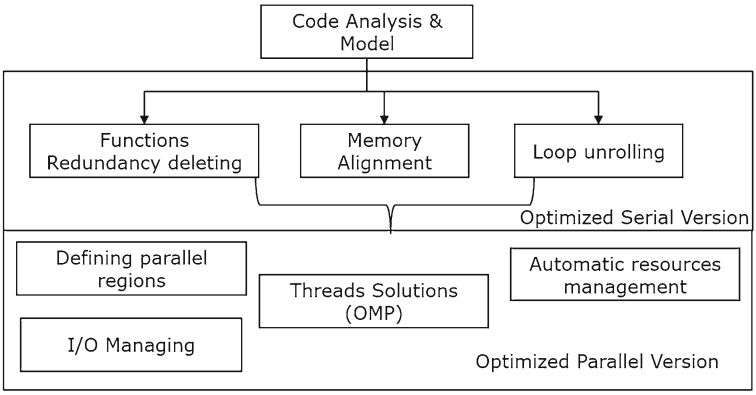

Fig. 8 Procedure to optimize SYMBOL Tool

For example, when comparing the results of Poland and Bulgaria, we see that both RegCap functions and the ABAC function has similar impact over the execution time. On the other hand, when the number of banks increase, the execution time in the RegCap function begins to drop, while the ABAC function grows considerably. as it is shown in Fig. 7 where Italy and Germany are considered. Summarizing we have that the RegCap function has a big impact on small input, while the ABAC function has a huge impact on large input data, and we have to take into account these findings when we start the optimization process.

3.2 Designing and Implementing the Optimization Techniques

The next step is to create an environment that can set up the best configuration for running the model by taking advantages of the multi-core architecture. It is important to notice that we have to take into account the overhead added to the parallelization, especially in the case of small input data, when they may give rise to longer execution time. Hence, we have decided to implement two different version of SYMBOL: an optimized serial (used for small input) and a parallel version (for large and medium input) (Fig. 8).

3.2.1 Creating a New Optimized Serial Version

In the serial version, we mainly focus on deleting the loop and functions' calls redundancies, managing the data structure using memory alignments, and applying the loop unrolling technique (Fig. 8) for deleting the computation redundancy. For example, Algorithm 1 shows a *C* source code for the RegCap function. This function is called across all iterations for each bank. This means that it is executed thousands or millions times depending on the number of defaults users wish to simulate. This RegCap function calculates the banks losses for banks i as was mentioned before in Eq. 1, using the data obtained in the previous module (ABAC function).

```
1  double RegCap(double x, double y, double k)
2  {
3   double a, ex1, ex2, TheR;
4   ex1 = 1-exp(-50*x);
5   ex2 = 1;
6   TheR = 0.12*ex1*ex2+0.24*((1-ex1)*ex2);
7   a = (k*gsl_cdf_ugaussian_P(pow(1-TheR, -0.5)*
        gsl_cdf_ugaussian_Pinv(x) + pow(TheR/(1-TheR), 0.5)*y)
8        - x*k)* pow(1-1.5*ba(x), -1)*1.06;
9   return(a);
10 }
11 /* Extract of the  Main SYMBOL Code  */
12 While(nDefaults < nTotDefaults) //
13 {
14  /* 1) Random Generator Function          */
15  /* 2) ABAC Function  */
16  for (i=0; i<nbanks; i++)
17  {
18       Baseloss = RegCap(data[i][0], market[b], LGD);
19  }
20  /* 3) Evaluating of a Default scenario   */
21  /* 4) Contagion Part of the Algorithm    */
22  if (hadDefault) //if there is a default
23             nDefaults++;
24 }
```

Algorithm 1 Original RegCap Function, C code Example

As it can be observed in Algorithm 1, the code calls the function in the line 19 of the algorithm, where it takes as input 3 variables: $data[b][0]$, which is the probability of default of the portfolio of obligors of each individual bank, $market[b]$ that is an array which was previously calculated using the random generator vector multiplied by the correlation matrix (calculated on ABAC function) and the loss given default (LGD) which is set to Basel regulation equal to 45 % (this code is an small example without including all the SYMBOL functionalities).

The optimization procedure moves from the analysis of the code to determine where dependencies exist. Then, we separate the part of the equation, which needs to be calculated in each iteration and the elements that can be calculated only once and then integrated into the RegCap functions. This leads to a new algorithm (see Algorithm 2), where, firstly, all elements without any crucial dependency are calculated. The new code integrates 4 global variables (A, B, C and D), which are the decomposition of the initial RegCap equation, and they are calculated for the same sample of banks of the simulation (line 9, Algorithm 2).

In other words, we have deleted the repetitiveness of computation, which was the main weakness in the original code. These values are calculated in the lines 9 through 17 of the new code, just before of the loop structure where the core of SYMBOL is run. This solution decreases considerably the execution time, but it also increases the memory allocation, due to the data of A, B, C and D must be maintained during all the execution. However, for the most demanding test performed (2773 banks), the amount of memory was 22 KB for each variable, which is not a relevant value considering that current computers that integrate more than 4GB of RAM memory. This is a clear example of recoding an algorithm in order to improve the effectiveness of the execution.

```
double RegCap2(int i, double y)
{
  return((LGD*gsl_cdf_ugaussian_P(*(A+i)+*(B+i)* y)-*(C+i))*(*(D+i)
      ));
}

/* Main SYMBOL Code  */
A, B, C, D, = (double *) malloc (sizeof(double)*nbanks);
ex2 = 1;
for(i=0;i< nbanks; i++)
{ex1=1-exp(-50*data[i][0]);
 TheR=0.12*ex1*ex2+0.24*((1-ex1)*ex2);
 ba=pow(0.11852-0.05478*log(data[i][0]), 2);
 *(A+i)= pow(1-TheR, -0.5)*gsl_cdf_ugaussian_Pinv(data[i][0]);
 *(B+i)=pow(TheR/(1-TheR), 0.5);
 *(C+i)=data[i][0]*LGD;
 *(D+i)=pow(1-1.5*ba, -1)*1.06;
}
While(nDefaults < nTotDefaults)
{
 /* 1) Random Generator Function */
 /* 2) ABAC Function */
 for (b=0; b<nbanks; b++)
      Baseloss = RegCap2(b, market[b]);

 /* 3) Evaluating of a Default scenario */
 /* 4) Contagion Part of the Algorithm */
 if (hadDefault) //if there is a default
    nDefaults++;
}
```

Algorithm 2 Recode RegCap function, Optimized version of SYMBOL

Then, we have tackle the efficient use of the data structures (i.e. the use of memory alignment), avoiding cases like struct vec double x, y, z, libraries data definitions, etc. SYMBOL was originally designed for using GSL library (Galassi et al. 2013), because a set of mathematical operations are used. However, it is better to avoid to define GSL data structures, if are not really needed. The main reason is that this kind of matrix structure integrates a set of data definition inside of the structure specifically six elements (Galassi et al. 2013). These definitions increase the memory allocation needs, which is an important key, when data on the code are aligned to take advantage of L1 and L2 cache memory.

Hence, we have translated the data structure of SYMBOL into memory array pointers. These allocations allow us to improve the spatial locality (Faria et al. 2013). Also, using this technique, we can reduce the cache misses and improve the execution time considerably.

Algorithm 3 shows the modifications done on SYMBOL code using the memory alignment and pointer arithmetic. This does not only allow us to improve the computational speed, but also, it creates the possibility to apply parallel strategies as it will be detailed in next section.

```
/* Code Using Gsl_ Matrix operation */
for (j = 0; j < nbanks; j++)
{
 tmp=0;
 for (i = 0; i < nbanks; i++)
 {
  tmp+=gsl_matrix_get(m,0,i)*gsl_matrix_get(c,j,i);
 }
 gsl_matrix_set (market, 0, j, tmp);
}

/* Code Using array of pointer */
for (j = 0; j < nbanks; j++)
{
 tmp=0;
 for (i = 0; i < nbanks; i++)
 {
  tmp+=*(m+i)* (*(c1+j*nbanks+i));
 }
 *(market+j)=tmp;
}
```

Algorithm 3 Comparison of transforming the code using pointer arrays for the ABAC function

Finally, we applied the loop unrolling technique in the most time consuming operation of SYMBOL, which is the ABAC function. The loop unrolling is a well-known code transformation for improving the application performance. This technique can improve from 10 to 30 % the execution time, depending on the impact of the loop inside the code (Davidson and Jinturkar 2001).

At this point, we have a new optimized SYMBOL serial version. Figure 9 summarizes the results obtained from different input data. The optimization techniques applied have considerably improved model's execution time from 60 % for the best case (Lithuania) and to more than 51 % for the worse case (Germany).

3.2.2 SYMBOL Parallel Version

Now that we have an optimized serial version, we move to implement a parallel version. In particular, we have assess if we can develop a version, which allows us to execute the model on a multicore cluster through:

- message passing interface (MPI) libraries (MPI 2012);
- parallel computing at machine level using a massive parallel programming technique through the (GPU) Graphics Processing Unit technology;
- implementing threads technology using OpenMP (OpenMP 2013) library.

It is worthwhile to notice that, SYMBOL presents two important data dependencies: the random generator sequence and the contagion part. This means that it has to execute module by module and each module needs to collect the information in order to calculate the next step as was illustrated on the Fig. 2.

We start to analyze Symbol using different parallel strategies. The first analysis was done using MPI libraries. In order to implement SYMBOL for MPI, we have to deal with a set of problems due to the communications overheads generated by communication message of each MPI processes. SYMBOL was designed using an iterative process in which default scenario occurs if some conditions hold. In other

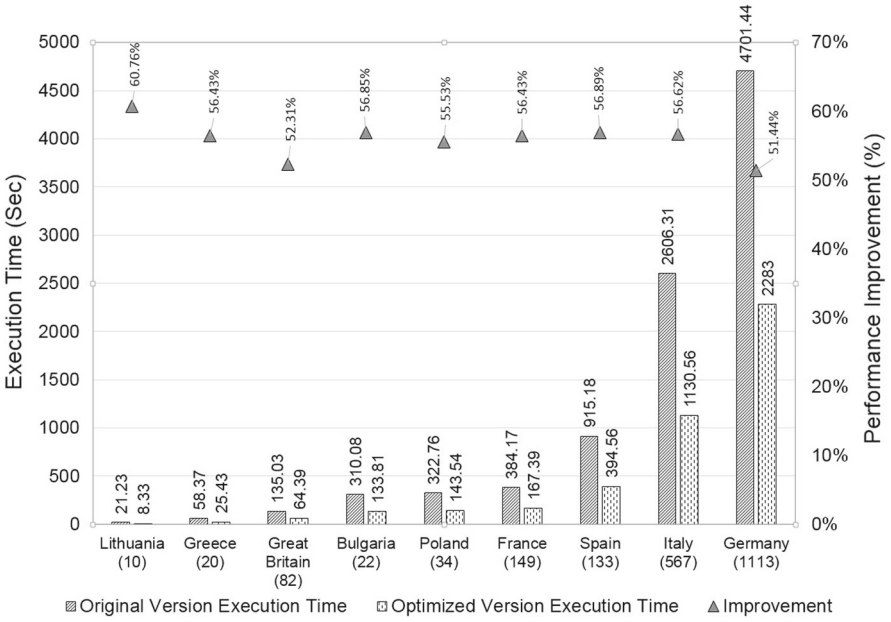

Fig. 9 Results of the new optimized and original serial version using 100.000 defaults scenarios

words, we cannot define a given number of iterations to find a specific number of defaults. Hence, if we split the workload into MPI processes, each process has to calculate its own defaults values and then transfer its partial results to the master process which would then group them together.

The second problem is related to the random number generation. SYMBOL generates a set of random numbers sequence and these randoms numbers will be multiplied with each column of the correlation matrix which represent a correlation among different banks assets. These numbers cannot be generated in different MPI processes, hence we generate a random sequence in serial and then communicate the random numbers previously generated to the MPI processes using a MPI Broadcast instruction.

This implies that in each iteration we have to communicate through the cluster network twice: first the random numbers from the master process to the workers and then the final results from the worker to the master to analyze a default scenario creating a huge overheads. These communication overheads, in all cases we have tested has leading to a poorer results in terms of execution time.

The second parallel parallel technique that can be used is the (GPU) Graphics Processing Unit technology. In this case, we have developed a version of SYMBOL using Opencl, which is one of the best known implementation for GPU (Stone et al. 2010).

Developing a code into the GPU is not an easy task, since we have have to avoid the communications between the main memory from the CPU (central processing unit) and the GPU memory. The communication between memories create a high overhead that GPU programmers are trying to solve it (Keckler et al. 2011). This

communications issue creates problems, because a sequence of random numbers has to be generated and then multiplied by the correlation matrix for all banks as we have observed on the flowchart of the Fig. 2.

The random numbers are generated into the CPU, then they are transfer to the GPU to calculate the the ABAC function. On the other hand, the RegCap function uses a set of *gsl* instructions which are not available for the OpenCl implementation.

This means that we need to transfer the results of the ABAC function from the GPU to the CPU to calculate the RegCap function. These two transfers from the main memory to the GPU and from the GPU to the CPU obviously create some delays that affects the execution time.

An example of the SYMBOL Kernel code generated to be used into the GPU can be observed in the Algorithm 4. In this kernel, we have translated the ABAC function whose inputs are the random vector ($*m\_gpu$), the correlation matrix ($*c\_gpu$) and the number of banks n.

```
1  kernel void symbol_kernel(_global const float *m_gpu, _global const
       float *c_gpu, _global float *tmp_gpu, const int n)
2  {
3  int j;
4  float tmp;
5  const int i=get_global_id(0);
6  /* Considering the threads identified less than the number of banks
       */
7  if (i<n)
8  {
9    tmp=0;
10   for(j=0;j<n;j++)
11   {
12      tmp+= *(m_gpu+j)*(*(c_gpu+i*n+j));
13   }
14   tmp_gpu[i]=tmp;
15 }
16 };
```

Algorithm 4 Symbol GPU Kernel

This modification allows considerable improvements with respect of the optimized serial version for large input data, but a worse execution for medium and small input (see Fig. 10). In particular Fig. 10 shows improvements of 12.88 times on speedup for a simulation using 2772 banks. In the case of Italy and Germany the increment of speed is around 1.5 and 4.6 respectively with respect to the serial optimized version.

In the case of small/medium size countries we get longer execution time. As an example we see that for a medium size country such as Spain the GPU version execution time is almost the double with respect to the serial version, for Great Britain is is almost five and a half time. This happens because of the communication transfer of the random vector between the main memory and the GPU and the transmission of the results from the GPU to the CPU. The overhead added by the GPU parallelization has a default overhead that must be covered by the computation in order to get improvements on the execution.[10]

[10] This experiment was done using a ATI Radeon HD 5770 GPU card with 1 GB of RAM and OpenCl Version 2.0.

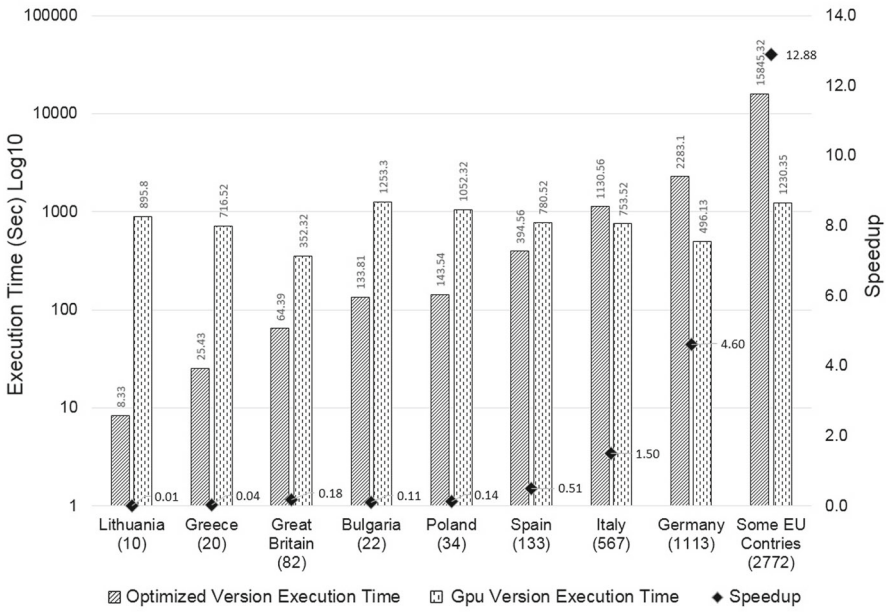

Fig. 10 SYMBOL with GPU Parallel results

Under this scenario, with the GPU, the next parallel development of SYMBOL is to use OpenMP (OpenMP 2013). Such platform is becoming more and more popular, because the number of cores inside the computer is increasing. Currently, we have systems where the number of cores are 4, 6, 8, etc., per processor and with multiple processors in each machine. These large number of cores would allow us to create a small high-speed environment with a shared memory space. Furthermore, OpenMP is a portable, scalable tool which allows programmers, using a simple interface, to create parallel region where the code is executed in parallel on multiple threads sharing data. However, once again, it is important to take care of data sharing between threads and to avoid the dependencies inside of the loop structures in order to not create deadlock scenarios. We refer to Fig. 11 to see what can go wrong in some cases. For this reason, we have looked for a solution allowing us to make a parallelization inside the machine with the aim of managing the overhead added by the parallel library.

Using OpenMP, we can create the random vector in serial and then other functions through parallel threads. The main advantage is that OpenMP uses a shared memory space that can be accessed by all threads.

Once medium/big input data are used we have a clear improvement by using the parallelization. It is worthwhile noticing that improvement depends on the number of threads opens for execution. As an example, we look at the case of Great Britain. Using 2 threads parallel version is faster than serial version while increasing the number of threads (e.g. 4, 6, 8, 10,…) the performance will not improve, because the overhead generated by threads is bigger than computation time of the workload. We have a similar behavior for Spain, where the execution time improves until 4 threads are opened (see Fig. 12).

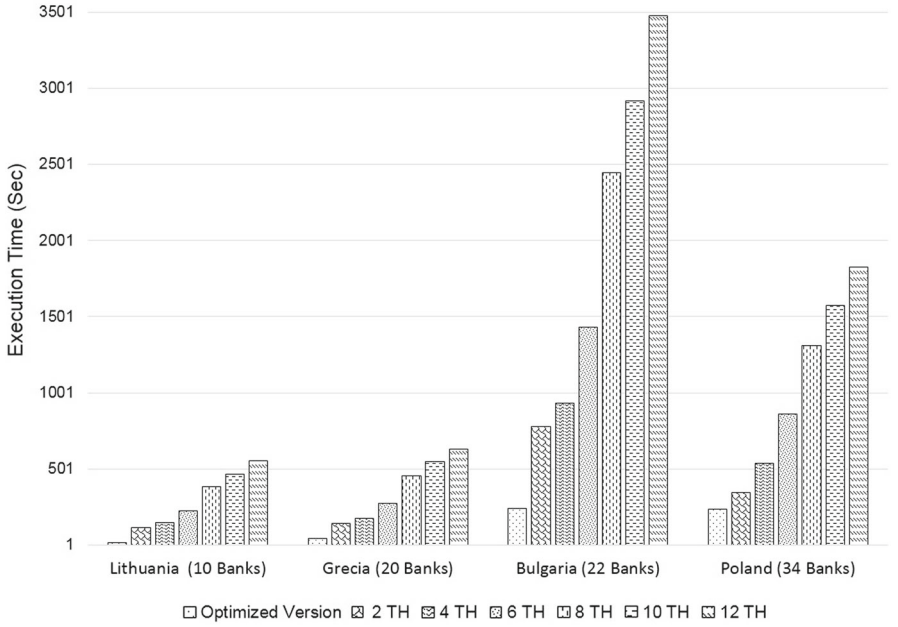

Fig. 11 Worst parallel results using SYMBOL and OpenmP

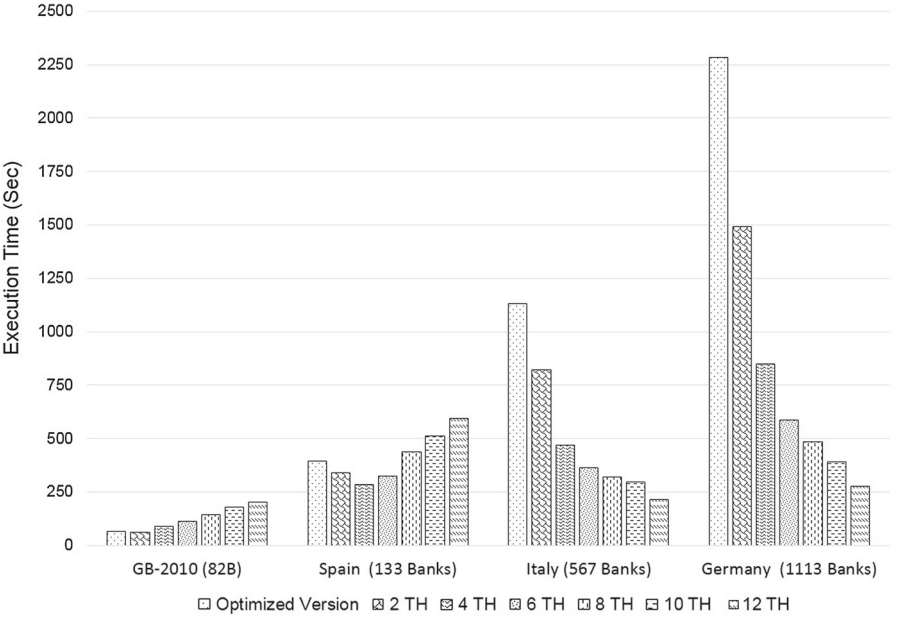

Fig. 12 Improving parallel execution of SYMBOL using OpenMP

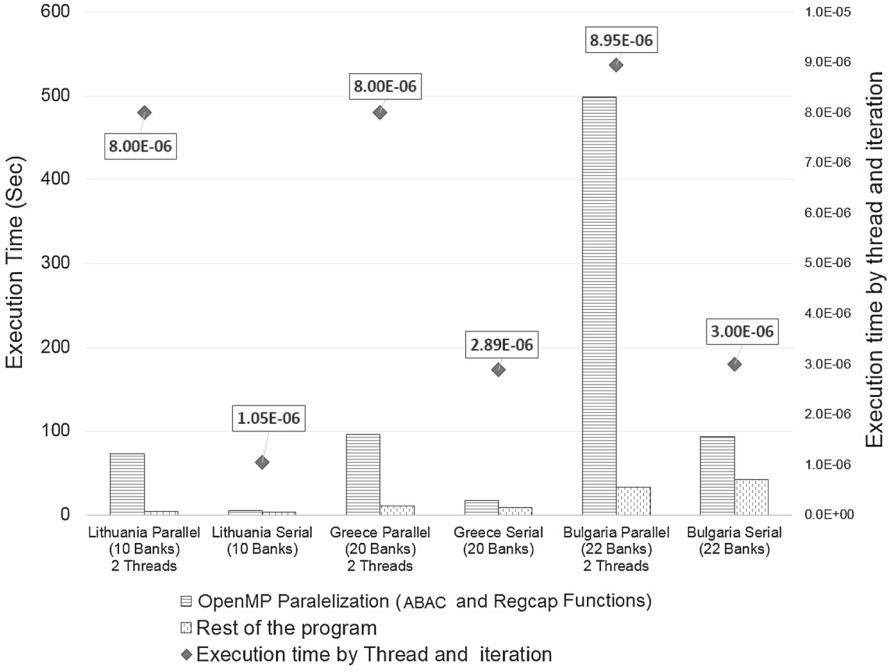

Fig. 13 Serial and OpenMP thread characterization for small input

Figure 12 also shows two additional examples (Italy and Germany) the parallelism is exploited through OpenMP. In both cases we have opened until 12 threads getting the shortest execution time. In the case of Germany, the execution time has dropped off from 38 minutes of the serial optimized version, to less than 5 minutes using the OpenMP version (a gain of around 87 %). However, it is important to understand that the number of threads will be limited by two main factors: number of physical cores on the machine and the computational workload of the input data.[11]

To address the issue of number of open threads, we need to determine the ideal number of threads according to the input data in a transparent manner. Hence, we have set up an automatic procedure that determines the ideal number of threads to be opened for the parallel version or it simply selects the optimized serial version. The ideal number of open threads is linked to the number of banks in the input dataset which leads us to evaluate the overhead created by the OpenMP implementation on each iteration inside SYMBOL.

An example of this characterization can be detailed in Fig. 13, where we have selected the worse cases for small inputs. For this analysis, we have divided the code in two main parts: where OpenMP primitives are used and the rest (Fig. 13). Because of the overhead added by the OpenMP library, serial execution outperforms OpemMP

[11] For these experiments, we have used a MAC machine with 2 processor Intel Xeon with 6 cores each and 64 GB of RAM memory. Hence, 12 is the maximum possible number of threads one can open for improving speed.

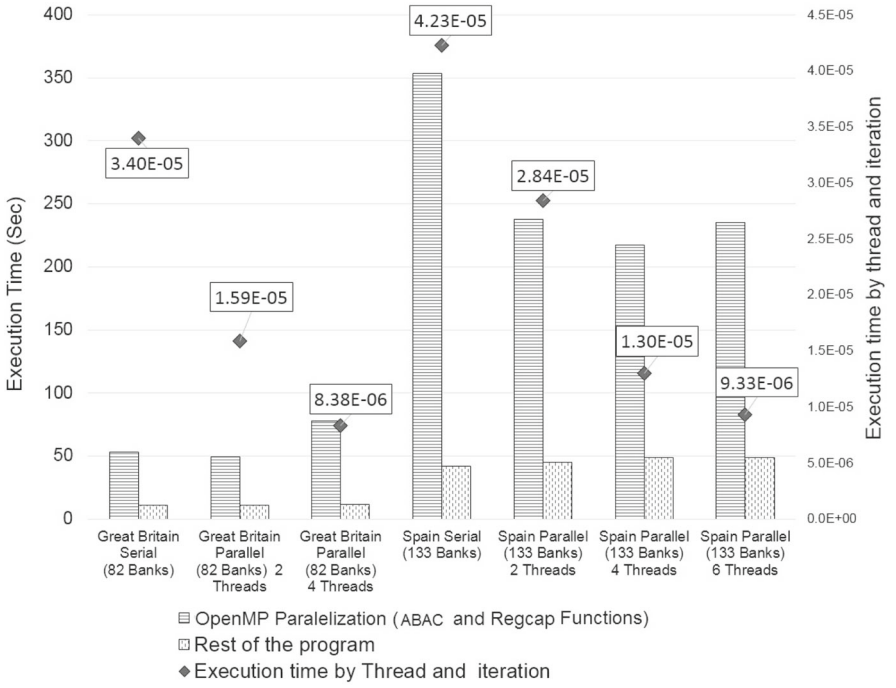

Fig. 14 Serial and OpenMP thread characterization for medium input

parallel version (using 2 threads) for these small input datasets (Lithuania, Greece and Bulgaria).

In particular, the average on the execution time by thread and iteration, are around 8.0E−6 and 9E−6 s for all inputs and the number of total iterations are 4.692.777, 5.874.684 and 2.787.7592 respectively (the stopping criterium is set at 100.000 defaults). On the other hand, average execution time for the serial version by iteration is lower than 3E−6. From these findings one may conclude that the minimum overhead in time by iteration that each thread must have around 9.0E−6 and 1.0E−5 s (these values have been defined as the minimum overhead threshold). If this overhead it is not covered, then we should use the serial optimized version in order to improve the performance.

Then we move to analyze medium input size dataset. In particular we focus on two cases Great Britain (82 banks) and Spain (133 banks). Figure 14 shows that, in both cases, OpenMP version outperforms the serial version. However, these improvements are achieved until an specific number of threads. For example in Great Britain the execution with 2 threads is better than serial, while if we execute with 4 threads the execution begin to become worse. For Spain, the best execution time is attained using 2 or 4 threads.

Using 4 threads for Great Britain and 6 threads for Spain the values of the execution time average by threads are lower than 10E−5, i.e. below the threshold of the minimum added overhead of the OpenMP implementation for this application. On the other hand

the execution time of Spain with 4 threads is around 1.3E−5 bigger than the minimum overhead. This situation corresponds to more or less 33 banks by thread. Hence we decided to set a threshold to 30 banks, which gives the minimum requirement in order to cover the overhead added by the OpenMP solution. Once again the number of threads is limited by the physical capacity of the multi-core architecture. In Sect. 4 this value has been tested with a set of different inputs.

Algorithm 5 details OpenMP implementation of parallel strategies for the ABAC function, in particular we have used OpenMP thread through the instruction *#pragma omp parallel* and the loop unrolling technique in order to reduce the looping overhead.

```
1
2 #pragma omp parallel for private (j,i) schedule(dynamic, 16)
      reduction (+:tmp) num_threads(num_threads)
3 for (j = 0; j < nbanks; j++)
4 {
5  tmp=0;
6  for (i = 0; i < nbanks; i=i+4)
7     tmp+= (*(c1+j*nbanks+i))* *(m+i)+(*(c1+j*nbanks+i+1))* *(m+i+1)
          +(*(c1+j*nbanks+i+2))* *(m+i+2)+(*(c1+j*nbanks+i+3))* *(m+i
          +3);
8  *(BankLoss+nDefaults*nbanks+j) = data[j][3]*RegCap2(j, tmp);
9  if (*(BankLoss+nDefaults*nbanks+j) > data[j][2])
10    hadDefault = 1;
11 }
```

Algorithm 5 ABAC and RegCap functions parallelized with OpenMP

Also, this algorithm has two private variables to avoid that other threads may modify their contents and we define an OpenMP scheduling clause "dynamic" and a data sharing attribute "reduction". The dynamic clause is used to balance the size of the threads workload in order to balance the execution, while the reduction attribute for the *tmp* variable is used to accumulate the partial threads results and at the end it will add all these partial values. Finally, we also have to declare the the number of threads to be opened.

Once we design how to implement the parallel techniques we are ready to create an automatic tool where everything we have deployed is embedded. This tool is compiled using pre-processor macros to control, using conditional compilation, which part of the code will be executed in serial or parallel (both for OpenMP and GPU versions). Moreover, in the case of parallel execution, we need to declare the maximum number of threads machine can support as detailed before. The user can also decide if GPU are going to be used. The parallel and serial versions are called using a batch file, and it will select the most appropiate version (Serial, OpenMP or GPU) according to the input analized. Also, it will open, in the case of OpenMP version, the ideal number of threads using the overhead analysis explained before. More specifically the tool is automatically executed and it follows the flowchart defined on Fig. 15.

Whenever the input dataset has less than 60 banks the execution will be performed in serial. Otherwise, it goes parallel and there is the possibility of selecting either the GPU version used for very large dataset (more than 1500 banks) to cope with the minimum overhead created by the GPU implementation. This number of banks is indeed an approximation of the time between the computational time for ABAC function and the communication transfer between memories. If the dataset has less

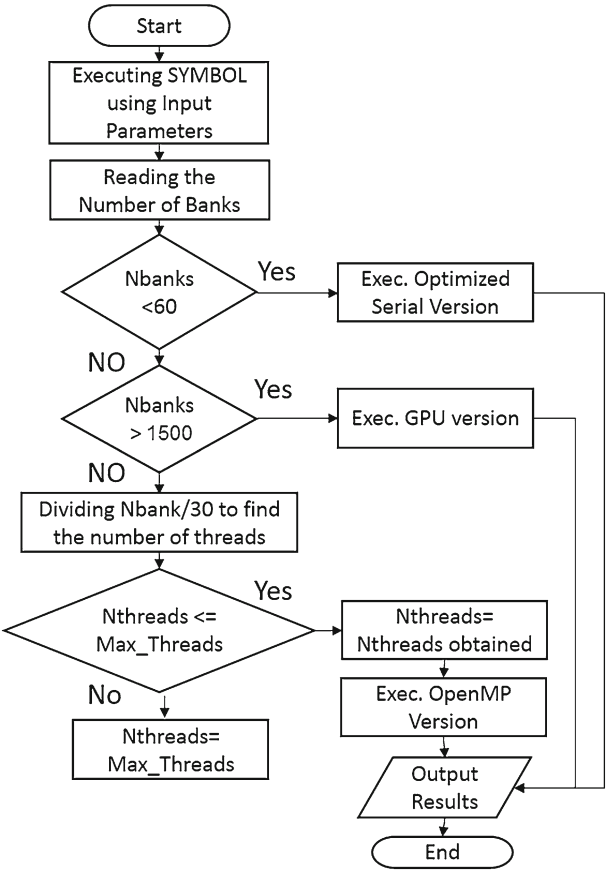

Fig. 15 Flowchart for the SYMBOL execution batch file

than 1500 banks, the OpenMP version is used. In this case the right number of open threads needs to be computed to hide the OpenMP overhead. If the number of threads is bigger than the number of cores of the machine, then the maximum capacity of the system is used. As reference example we consider Germany with around 1100 banks. In this case the right number of open threads is the minimum between $\min(37 \sim \frac{1100}{30}$ and the number of cores $\#cores$).

4 Experimental Evaluation and Optimization Results

4.1 Testing and Data Verification of SYMBOL

The tool (hence all new versions of the SYMBOL model) has been tested with a strict procedure, comparing it with the original version using a set of different input data. Simulations' results are 100 % in agreement for all versions (original, serial, GPU and OpenMP). To test the tool, we have used a MAC machine composed by two processor

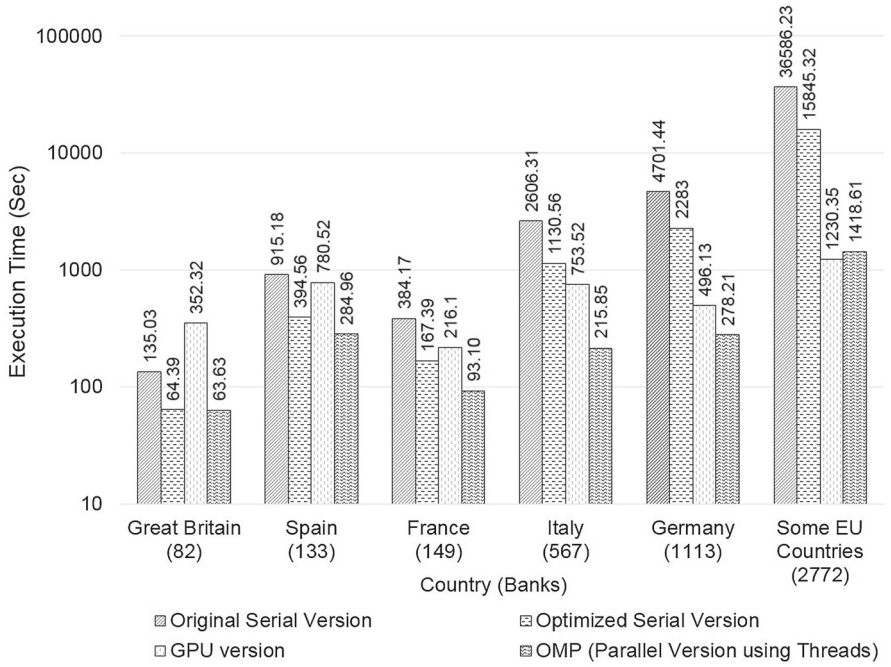

Fig. 16 SYMBOL executions using the optimized serial, GPU and OpeMP parallel version

of 2.93 Ghz 6 Core Intel Xeon, 64 GB 1333 Mhz DDR3 memory RAM, Mac OS X lion 10.7.5 operative system, and gcc 4.6.2 compiler and a video card with GPU technology Ati Radeon HD 5770.

4.2 Final Comments on the Results

The tool developed to improve performances of the SYMBOL model addresses the following aspects: execution time and scalability with respect to the stopping criterium. Figure 16 shows execution time obtained for 100.000 defaults using the new SYMBOL tool.

The tool selects the best execution version depending on the size of the input dataset. Possible alternative are: serial, GPU or OpenMP version.

Results reported in Fig. 16 refer to: the original serial version, the optimized serial version, the OpenMP and the GPU version for a selection of possible input datasets (Great Britain, Spain, Germany and EU). If we focus on Great Britain we observe that the lower execution time for is 63.33 using the OpenMP version using 2 threads (more or less 47 % of the original execution time, i.e. an improvement around 53 %). For Spain we have an improvement of 68.88 % using the OpenMP with 4 threads, (using the GPU version we got a worse execution due to the overhead of the GPU implementation). GPU turns to be more efficient for Italy (71 % reduction) and Germany (89 %). Nevertheless using OpenMP version with 12 threads reductions increase up to 91.71 %

Table 1 Summary of SYMBOL execution time with 100.000 defaults

Data Input (Nbanks)	Version (Th.Opened)	Original Exec.(s)	Optimized Exec.(s)	Parallel Exec(s)	I/O oper. Exec.(s)
Spain (133)	OpenMP(4)	915.46	394.56	275.64	5.32
France (149)	OpenMP(4)	384.17	167.39	93.11	6.12
Italy (567)	OpenMP(12)	2606.31	1130.56	218.49	18.85
Germany (1113)	OpenMP(12)	4701.44	2283.01	276.42	38.14
Some EU Countries (2727)	GPU	36586.23	15845.32	1230.35	99.38

Table 2 Performance evaluation of SYMBOL versions with 100.000 defaults

Data Input (Nbanks)	Version (Th.Opened)	Speedup Orig./Parallel	Speedup Optim./Parallel	Efficiency (%) Optim./Parallel
Spain (133)	OpenMP(4)	3.37	1.44	36.00
France (149)	OpenMP(4)	4.35	1.85	46.35
Italy (567)	OpenMP(12)	12.96	5.57	46.40
Germany (1113)	OpenMP(12)	19.57	9.42	78.51
Some EU Countries (2727)	GPU	29.73	12.88	107.03

for Italy and 94.08 % for Germany. Finally, for the whole EU sample (2772 banks) GPU version outperforms all others with a 96.63 % reduction, little bit better than OpenMP with 12 threads (the maximum available on our architecture). In terms of speed up we get 29.73 for GPU and 25.79 for OpenMP.[12]

We recall that results of speedups and execution times as presented in Fig. 16 include the input/output (I/O) operations, which are not considered for the current optimization.

To properly assess the effective execution time on multi-core architecture, we need to estimate the impact of the I/O operations on the efficiency and the speedup.

Table 1 shows the time for the I/O operations: in particular we see that the impact in Spain is around 2 %, while in Germany is 14 % of the total execution time. Then, we have to subtract this I/O time in order to know the CPU time of the SYMBOL tool executions.

Overall performance, in terms of speedup and computational efficiency, evaluations are shown in Table 2 where we present the serial time divided by the parallel time. The speedup results are shown in Column 3 (parallel versus original serial) and 4 (parallel versus optimized serial). We remind that ideal speedup is achieved, when its value is equal to the number of parallel resources. One can see in Column 3, the speedup values are bigger than the amount of parallel resources used. The reason being that the parallel version was designed starting from the new optimized SYMBOL version

[12] The speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

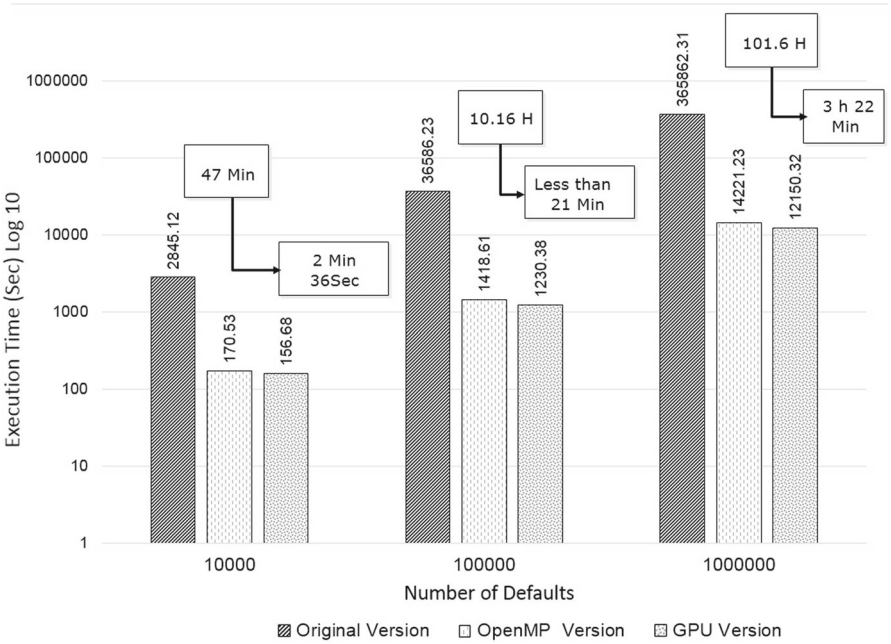

Fig. 17 SYMBOL scalability analysis

which already gives a benefit (larger than 50 %) with respect to the original serial version (see Fig. 9).

In terms of which version is selected (in a transparent manner) by the tool, from Tables 1 and 2 we see that in the case of Spain, France, Italy and Germany, OpenMP is selected with the optimal number threads depending on the number of banks. These open threads can generate additional overheads that can worsen the linearity of speedup.

Nevertheless, if we increase the input data size and we maintain the same computational resources, we can achieve almost a linear speedup, e.g. "Some EU countries" in Table 2. In this case, GPU version has been used, where the speedup obtained was around 12.88 with a system efficiency of 107.03 %[13] using a machine with 12 cores.

The second aspect we want to focus on is scalability of the tool with respect to number of default scenarios. The exercise considers the increase of number of default scenarios from one hundred thousand to one million and the results are summarized in Fig. 17.

It can be seen that considerable improvements are achieved by both OpenMP and GPU version when a large input dataset is used and a large number of default scenarios are considered: best result shows a 96.67 % reduction for execution time (i.e. few hours instead of several days; and speedup around 30 for the GPU version).

[13] In this case, we have obtained an efficiency greater than 100 % due to we have used the GPU card cores and not the CPU cores to execute this input.

5 Conclusions

In this paper, we have presented methods and strategies to adapt an existing serial code of the SYMBOL model to multi-core architecture and GPU technology (both serial and parallel optimizations are indeed considered).

The analisys moves from highlighting the main characteristics of the model to find which are, in the original codes, the most relevant points affecting performance. Then, we have designed a method to create the optimization plan (easily applicable to other model having similar characteristics) in order to take advantage of latest IT developments.

We looked for an automatic tool which sets up its working framework depending on the input datasets and on the available resources. In particular it detects (in an automatic and transparent manner) which version to use either serial or parallel and in the latter case it selects number of threads needed to get a faster execution using the input information and it decides whether to use GPU or OpenMP.

Results of the new tool have been experimentally evaluated using several datasets, which differ in terms of number of banks and in terms of underlying data structure. They clearly shows the effectiveness of the new optimization, both for the new serial and for the parallel (OpenMP and GPU) codes. All versions reduce significantly the execution time going from 50 % reduction (serial version) up to about 96 % reduction (parallel optimization).

Finally, but not less important, the tool is very effective when we face the problem of scaling up the size of the input dataset as well as the number of default scenarios needed to have less uncertainty in the results.

References

Basel Committee on Banking Supervision. (2005). *An explanatory note on the basel II IRB risk weight functions*. http://www.bis.org/bcbs/irbriskweight.pdf.

Basel Committee on Banking Supervision. (2006). *International convergence of capital measurement and capital standards*. http://www.bis.org/publ/bcbs128.pdf.

Basel Committee on Banking Supervision. (2010) (rev 2011). *A global regulatory framework for more resilient banks and banking systems*. http://www.bis.org/publ/bcbs189.pdf.

Davidson, J. W. & Jinturkar, S. (2001). *An aggressive approach to loop unrolling*, technical report, University of Virginia, USA.

De Lisa, R., Zedda, S., Vallascas, F., Campolongo, F., & Marchesi, M. (2011). Modelling deposit insurance scheme losses in a Basel 2 framework. *Journal of Financial Services Research*, *40*(3), 123–141.

De Rose, C., Fernandes, P., Lima, A., Sales, A., & Webber (2011). Exploiting multi-core architectures in clusters for enhancing the performance of the parallel bootstrap simulation algorithm. *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)* (pp. 1442–1451).

Faria, N., Silva, R., & Sobral, J. (2013). Impact of data structure layout on performance. *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 117–120), Ireland.

Galassi, M., Davies, J., Theiler, J., Brian, G., Jungman, G., Alken, P., Booth, M., & Rossi, F. (2013). *GNU scientic library*, reference manual. http://www.gnu.org/software/gsl/manual/gsl-ref.pdf.

James, C. (1991). The loss realized in bank failures. *Journal of Finance*, *46*, 1223–1242.

Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., & Glasco, D. (2011). GPUs and the future of parallel computing. *IEEE Micro*, *31*(5), 7–17.

Merton, R. C. (1974). On the pricing of corporate debt: the risk structure of interest rates. *Journal of Finance*, *29*, 449–470.

Message Passing Interface Forum. (2012). *MPI: A Message-Passing Interface Standard Version 3.0*, technical report. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

Michailidis, P., & Margaritis, K. (2012). Efficient multi-core computations in computational statistics and econometrics. *Proceedings of the IEEE 15th International Conference on Computational Science and Engineering (CSE)* (pp. 267–274).

Mistrulli, P. E. (2007). *Assessing financial contagion in the interbank market: maximum entropy versus observed interbank lending patterns*, Bank of Italy Working Papers No. 641.

OpenMP Architecture Review Board. (2013). *OpenMP Application Program Interface*. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

Sironi, A., & Zazzara, C. (2004). Applying credit risk models to deposit insurance pricing: Empirical evidence from the Italian banking system. *Journal of International Banking Regulation*, *6*(1), 10–32.

Stone, J. E., Gohara, D., & Shi, Guochun. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Journal in Computing in Science Engineering*, *12*(13), 66–73.

Upper, C., & Worms, A. (2004). Estimating bilateral exposures in the German interbank market: Is there danger of contagion? *European Economic Review*, *8*, 827–849.

Vasicek, O. A. (2002). *Loan portfolio value*, Risk. http://www.risk.net/data/Pay_per_view/risk/technical/2002/1202_loan.pdf.

Zedda, S., Cannas, G., Galliani, C., De Lisa, R. (2012). *The role of contagion in financial crises: An uncertainty test on interbank patterns*, EUR report 25287. ISSN 1831-9424, ISBN 978-92-79-23849-9. http://publications.jrc.ec.europa.eu/repository/bitstream/111111111/25695/1/lbna25287enn.pdf.